The International Journal of Time-Critical Computing Systems, 17, 65–86 (1999)

Stabilizing Pre-Run-Time Schedules With the Help of Grace Time

ANTÓNIO PESSOA MAGALHÃES apmag@fe.up.pt
SAIC - DEMEGI Faculty of Engineering, University of Porto - PORTUGAL

JOÃO GABRIEL SILVA jgabriel@dei.uc.pt
GSC - DEI Faculty of Sciences and Technology, University of Coimbra - PORTUGAL

Abstract. This paper discusses the stability of a feasible pre-run-time schedule for a transient overload introduced by processes re-execution during an error recovery action. It shows that the stability of a schedule strictly tuned to meet hard deadlines is very small, invalidating thus backward error recovery. However, the stability of the schedule always increases when a real-time process is considered as having a nominal and a hard deadline separated by a non-zero grace time. This is true for sets of processes having arbitrary precedence and exclusion constraints, and executed on a single or multiprocessor based architecture. Grace time is not just the key element for the realistic estimation of the timing constraints of real-time error processing techniques. It also allows backward error recovery to be included in very efficient pre-run-time scheduled systems when the conditions stated in this paper are satisfied. This is a very important conclusion, as it shows that fault-tolerant hard real-time systems do not have to be extremely expensive and complex.

Keywords: Real-time systems, pre-run-time scheduling stability, grace time, fault-tolerance.

1. Introduction

Real-time literature shows a tendency for deriving the timing constraints of a real-time service from its timeliness. That is, from a function that maps the merit of a service to its delivery time (Bond, Seaton, Veríssimo and Waddington, 1991). The timeliness of a service is always established in some particular metrics according to service importance for a real-time application. Usually, it takes the form of a *time-value function* (Jensen, Locke and Tokuda, 1985), *time-utility function* (Burns, 1991) or, more generically, *cost function* (Shin, Krishna and Lee, 1985).

Using this approach, a real-time designer can identify a set of time milestones attached to a service having several and specific goals. Each milestone represents a particular deadline and has a suitable semantics revelling specific concerns and consequences if missed (Bond, Seaton, Veríssimo and Waddington, 1991; Burns, 1991; Jensen, 1993; Jensen, 1994; Geith and Schwan, 1993; Kligerman and Stoyenko, 1986; Laplante, 1993; Ramamritham, 1993). Unfortunately, classical scheduling theory (Cheng, Stankovic and Ramamrithan, 1987; Audsley and Burns, 1992) typically takes single deadline processes and it does not seem to provide a direct support for this approach. On the other hand, *best-effort* scheduling algorithms based on processes' time value functions (Locke, 1986) are not the optimal solution, since they tend to be unpredictable during transient overloads (Burns and Fohler, 1991). Scheduling algorithms that can guarantee various performance or safety levels (Stankovic, Spuri, Di Natale and Buttazzo, 1995) are thus required.

However, as far as we know, real-time scheduling algorithms explicitly supporting processes having more than one deadline do not seem to exist (Magalhães, 1996). As a consequence, most real-time scientists tend to design their systems considering the hard deadline of each critical process (Laplante, 1993) (Stankovic and Ramamrithan, 1993). Yet, the *stability* of a system designed in this way may be very poor, in the sense that a hard deadline can be easily missed at the impact of a minor non-deterministic event. Thus, applications strictly tuned to meet hard deadlines have to assume a fault-free environment or a very complex and expensive structural redundancy providing error masking. This is particularly notorious when high processor utilisation is a major concern and hard deadlines tend to be satisfied only by a short margin, even in the absence of a fault.

Recently, we proposed a unifying approach that intents to contribute to change this view (Magalhães, Rela and Silva, 1996). It departs from the establishment of the timeliness of a real-time service according to a *cost function*, concluding that a real-time process always presents a *nominal deadline*, and may exhibit a *hard deadline*. The nominal deadline defines the maximum completion time that still guarantees the intended effectiveness of the process for the application; the hard deadline establishes the maximum completion time that prevents a catastrophic timing failure.

It is universally agreed that the first aim of every real-time application is the delivery of some beneficial service, although safety has always to be guaranteed (Bond, Seaton, Veríssimo and Waddington, 1991; Leveson, 1986). Thus, each process being part of a real-time system must be scheduled to meet its nominal deadline under normal circumstances, and not to miss its hard deadline in any case. Since the cost associated to a real-time service increases with its delivery time (Shin, Krishna and Lee, 1985) one finds that the nominal deadline of a process is more stringent than its hard deadline. From here, an important conclusion emerges:

If, due to a fault, a process misses its nominal deadline, a catastrophic timing failure will not immediately occur, but only later; namely, when the time interval separating the nominal and the hard deadline of the process exhausts. Such a time interval is called *grace time*—Figure 1. This grace time definition closely follows Kirrmann (1987).

Systems designed around nominal and hard deadlines are more stable than those strictly tuned to meet hard deadlines. This is because, in the first case, processes are allowed to miss their nominal deadlines by the corresponding grace time when the controller suffers the impact of a non-deterministic event. Namely, a transient overload introduced by processes executing for a time greater than the expected. This is very important in the context of fault-tolerant computing. Particularly when low cost solutions are required, as it is usually the case (Avizienis, 1997):

When processes are scheduled to satisfy their nominal deadlines, *backward error recovery* (Rennels, 1984; Laprie, 1991; Somani and Vaidya, 1997; Ziv and Bruck, 1997) may become a viable technique. As long as the recovery time is lower than the grace time of the affected processes, system safety does not suffer. Thus, only very hard real-time systems—i.e., those that include one or more processes having a very small grace time—can not use time-consuming error processing techniques. But these systems, while very important, are just a small minority. Case studies show that grace time can span from just a few milliseconds

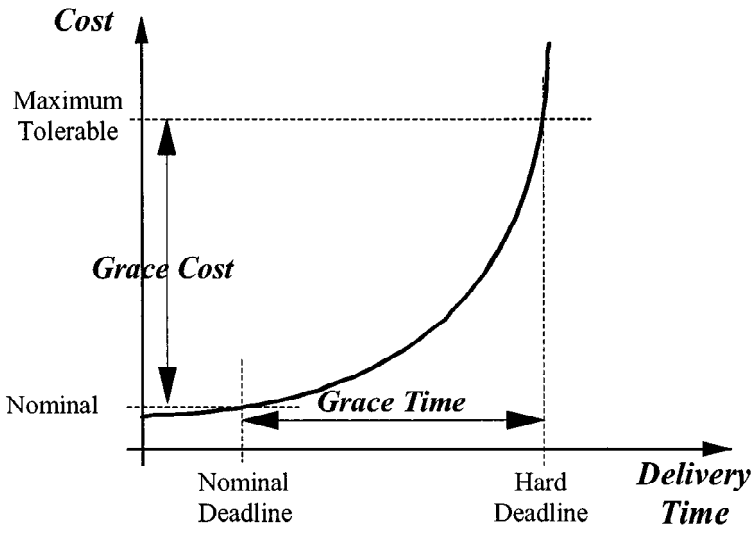

Figure 1. Deadlines establishment and associated grace time (Magalhães, Rela and Silva, 1996).

for very critical control loops, to tens of seconds for supervisory control (Kirrmann, 1987; Magalhães, 1995).

It is worth noting that backward error recovery has its own limitations. Namely, it can only be applied to reversible actions—i.e., computations. Backward error recovery can not remove error masking sensors and actuators from hard real-time applications. However, replicated I/O devices (Kopetz and Veríssimo, 1993, Iyengar, Prasad and Min, 1995) are not the main contribution to the great complexity and the enormous price of actual fault-tolerant real-time systems.

The motivation for this paper is thus the notion that grace time is the key element for designing low-cost, yet highly reliable and efficient, hard real-time systems. Pre-run-time algorithms are considered because they are the most suitable for hard real-time applications (Kopetz, 1995; Xu and Parnas, 1991) but are incapable of dealing with unpredictable environmental or operational changes. Consequently, pre-run-time scheduled systems typically require error masking (Carlow, 1984; Kopetz, 1989; Shepard and Cagné, 1991; Driscoll and Hoyme, 1992; Carpenter, Driscoll, Hoyme and Carciofini, 1994). However, as shown in this paper, this is not an absolute requirement when grace time is considered. The paper is organised as follows:

Section 2 presents a scheduling model that reflects our nominal and hard deadlines concepts. Section 3 quantifies the stability of a feasible pre-run-time schedule for a transient overload introduced by process re-execution during an error recovery action. This quantification, covering single and multiprocessor systems, is done in two contexts: ignoring and considering grace time. The results are compared, showing that the stability of a schedule always increases in the second case. The quantification of the maximum time redundancy

that can be included in a computer system based on pre-run-time scheduling is later provided. Section 4 presents a short note discussing the returning of a pre-run-time schedule to normal conditions after suffering the impact of a transient overload. Section 5 concludes the paper summarising the most important conclusions.

2. Scheduling Model

The assurance that the timing constraints of all the processes running on a real-time computer are satisfied, requires to postulate a *scheduling model*. Usually, a scheduling model includes a *load model* and a *fault model* (Kopetz and Veríssimo, 1993). The load model, which ignores faults, specifies available processors, executing processes, and a criterion to define a schedule as feasible. The fault model defines the types and frequency of faults that the system must be capable of handling. Since the purpose of this paper is to derive the conditions that allow backward error recovery in a pre-run-time scheduled real-time system, it departs from a *load model* and a *fault-tolerance model*. From here, it is possible to derive the maximum time redundancy that can be used for error processing and the maximum frequency of faults.

The load model only differs from traditional ones in the sense that it defines a nominal and a hard deadline attached to each real-time process. Actually, it is very similar to the models presented in (Xu and Parnas, 1990) and (Shepard and Cagné, 1991) if the grace time of all processes is equal to zero. This means that our theory does not refuse the traditional view of real-time processes having a single deadline, but only makes it a particular case of a new and broader approach.

The fault-tolerance model is also very general. It is based in backward error recovery and derives from the specialised literature (Laprie, Arlat, Béounes and Kanoun, 1990; Laprie, 1991). Central to this model is a *stability criterion*. That is, a criterion that defines a schedule as feasible in the presence of a fault.

2.1. Load Model

The paper considers the execution of n processes on m processors. The set of processors is defined as $V = \{V_1, V_2, \ldots, V_m\}$. Each element of the set represents a unique processor. The m processors can be viewed as having identical or arbitrary processing capabilities.

The set $P = \{P_1, P_2, \ldots, P_n\}$ represents the n processes to be scheduled by a pre-run-time scheduling algorithm. For a processor $V_i \in V$, the subset $P(V_i) \subseteq P$ represents the set of processes allocated to V_i. Each process allocates to a single processor. Both periodic and sporadic processes are considered in P. A periodic process, P_p, is characterised by a set of five parameters: $(T_p, C_p, ND_p, HD_p, Q_p)$. T_p is the period of the process; C_p is the upper bound on its execution time; ND_p and HD_p are, respectively, the nominal and the hard deadline of P_p; Q_p is the processor on which P_p executes. For all periodic processes it is assumed that $0 \leq C_p \leq ND_p \leq T_p$ and $ND_p \leq HD_p$. The grace time of a periodic process P_p is, by definition, given by $GT_p = HD_p - ND_p$.

In a similar way, a sporadic process P_s is characterised by the set $(T'_s, C_s, ND_s, HD_s, Q_s)$. T'_s represents the minimum time interval between two successive requests of P_s. The parameters C_s, ND_s, HD_s and Q_s keep the meaning declared for periodic processes. However, pre-run-time scheduling always requires replacing all sporadic processes by polling periodics (Kopetz, 1991). A method for making such a transformation for dual deadline processes is thus required.

Following (Mok, 1984), and while viewing a "standard" deadline as a nominal one, the transformation of a sporadic process P_s into a periodic process P_p is done by observing the following conditions:

$$C_p = C_s; \tag{1}$$

$$ND_s \geq ND_p \geq C_s; \tag{2}$$

$$T_p \leq \min(ND_s - ND_p + 1, T'_s). \tag{3}$$

Two extra conditions are required for adapting Mok's theory to dual deadline processes running on a multiprocessor system:

$$Q_p = Q_s; \tag{4}$$

$$HD_p - ND_p \leq HD_s - ND_s. \tag{5}$$

Condition (4) keeps P_s allocated to Q_s; condition (5) preserves the grace time of P_s.

All processes in P can thus be seen as periodic and requesting execution at the beginning of the period, as it is usual in a pre-run-time scheduling environment. Therefore, any process $P_i \in P$ can be characterised by the set $(T_i, C_i, ND_i, HD_i, Q_i)$. All these parameters are non-negative integers given in multiples of the basic time unit of the system.

The load model also considers *precedence* and *exclusion* constraints between processes. A precedence constraint declares that a process P_i producing data to a process P_j must be scheduled to completion before P_j starts execution. A precedence constraint between two processes P_i and P_j denotes as $(P_i \prec P_j)$, meaning that P_i *precedes* P_j. Precedence constraints can exist between processes scheduled on the same or on different processors. The set of precedence constraints in P denotes as $PRE = \{(P_i, P_j) \mid P_i, P_j \in P \wedge (P_i \prec P_j)\}$.

An exclusion constraint between two processes P_i and P_j is denoted as $(P_i \otimes P_j)$, meaning that if P_i has started execution and it is not yet finished, then P_j cannot be started. Exclusion constraints can be established between processes scheduled on the same or on different processors and are symmetrical: $(P_i \otimes P_j) = (P_j \otimes P_i)$. The set of exclusion constraints in P denotes as $EXC = \{(P_i, P_j) \mid P_i, P_j \in P \wedge (P_i \otimes P_j)\}$.

Since the aim of a pre-run-time scheduling algorithm is to generate a *feasible* schedule, a feasibility criterion is required. In here, a schedule is declared as feasible if, in the absence of a fault, it guarantees that:

- All the precedence and exclusion constraints between processes are respected;
- No process starts executing before requesting execution;
- The completion time of every process is lesser than or equal to its nominal deadline.

2.2. Fault-Tolerance Model

The fault-tolerance model assumes a real-time system using backward error recovery (Laprie, Arlat, Béounes and Kanoun, 1990). In the value domain, the system is supposed to tolerate any permanent or transient fault originated from hardware or software. For achieving this, every time an error is detected the system is brought back into an error free state occupied prior to error occurrence—a *recovery point* (Nelson and Carroll, 1987; Kopetz and Veríssimo, 1993)—restarting execution from there. All fault treatment actions required in this scenario are supposed to be taken. Acceptance tests and checkpoint state storage are supposed to be regularly performed by the executing processes. The time required to perform these actions is included in processes execution times. Also assumed is that a process must pass an acceptance test before completing execution.

On a first approach, the model assumes that the processing of an error detected during the execution of a process P_i only leads P_i to rollback its execution. In this case, P_i increases its execution time by a margin ΔC_i. In the presence of a fault, P_i's execution time is thus equal to $C_i + \Delta C_i$, where ΔC_i denotes the error processing latency. On a second approach, the model assumes that an error processing action requires multiple processes to rollback their executions. This is a realistic view for preemptive schedules, where the running process as well as all the preempted processes must rollback execution every time an error is detected. This two step approach is because the analysis required in the second case is more complex than in the first, but it is easily driven from there.

Only two restrictions are assumed for the fault-tolerance model. First, faults are considered rare enough so the effects of two consecutive error recovery actions do not overlap. Second, no faults impacting the system introduce an error in more than one processor.

To define a schedule as *feasible* in the context of a fault, a special "feasibility criterion" is required: a *stability criterion*. In here, a schedule is *stable* if, in the presence of a fault, it guarantees the precedence, exclusion, start time and hard deadlines constraints of all the processes. This *stability criterion* closely follows the one stated in (Sha, Lehoczky and Rajkumar, 1986) for prioritised preemptive scheduling. However, our calculations of the stability margin of a real-time system will be very different from those presented in (Lehoczky and Ramos-Thuel, 1992), (Ramos-Thuel and Lehoczky, 1993) and other papers devoted to fixed priority scheduling. Fixed priority scheduling considers that every process requests its first execution at system start-up, and that a process P_i is allowed to preempt processes $P_{i+1}, P_{i+2}, \ldots P_n$, which have no permission to preempt P_i (Liu and Layland, 1973). These constraints are not usually present in pre-run-time scheduling. As a direct and major consequence, it suffices to consider the layout of a fixed priority schedule in the time interval $[0, T_n]$ to derive system stability, while, in the general case, pre-run-time scheduling requires the observation of the interval $[0, \mathrm{LCM}(T_1, T_2, \ldots, T_n)]$.

3. Stability Analysis of Pre-Run-Time Schedules

Due to their inflexibility, pre-run-time scheduled systems can hardly support non-deterministic events such as transient overloads caused by backward error recovery. By handling a

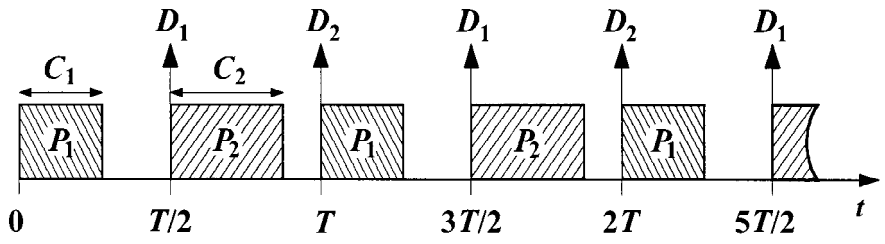

Figure 2. A pre-run-time schedule.

transient overload we mean to exhibit the necessary stability for guarantying processes hard deadlines during the overload manifestation, and later returning to nominal conditions.

Consider figure 2. It depicts a feasible schedule for a single processor real-time application consisting of two periodic processes, P_1 and P_2. Let $T_1 = T_2 = T$ and $C_1 < C_2 < T/2$. According to the usual approach, assume that both processes have a single deadline, such that $D_1 = D_2 = T/2$. Also assume that P_1 and P_2 request execution for every time $t = kT$ and $t = (2k + 1) * T/2$, respectively, where k is an integer greater than or equal to zero.

Observing figure 2 one infers that an overload thatmakes P_1 to increase its execution time by more than $T/2 - C_1$ makes P_1 to exhibit a timing failure. In that case, the start time of P_2 is delayed. In a similar way, an overload that makes P_2 to enlarge its execution time by more than $T/2 - C_2$ also leads to a timing failure. It can thus be stated that the *stability margin* of the schedule for an overload introduced by the execution of P_1 and P_2 is given by $T/2 - C_1$ and $T/2 - C_2$, respectively. If any process P_1 or P_2 enlarges its execution by more than $(T/2 - C_1) + (T/2 - C_2)$, then both processes miss their deadlines. Although this is a very simple scenario, it provides two important conclusions:

- Expected idle times are central to the quantification of the stability of a pre-run-time schedule;
- A single transient overload can make a set of processes miss their deadlines in a row.

These conclusions are enhanced if one changes the scenario depicted in figure 1 such that $C_1 = C_2 = T/2$. In that case, although the schedule is still feasible, it can be defined as *supercritical*, in the sense that its stability margin reduces to zero for any process. Any minor overload can start a chain reaction from where no process ever meets its deadline: the "*domino effect*" (Buttazzo, Spuri and Sensini, 1995). However, if D_1 and D_2 are viewed as the nominal deadlines of P_1 and P_2, respectively, and if both processes have a grace time greater than zero, then the stability margin of the schedule is greater than zero, even if $C_1 = C_2 = T/2$. This is true for any feasible schedule, as it proves trivially:

THEOREM 1 *Any feasible schedule of processes having grace times greater than zero has a stability margin greater than zero.*

Proof: Assume that, in the absence of an overload, a schedule guarantees the nominal deadline of every process. That is, every process P_i completes execution by a time $c(P_i)$ such

that $c(P_i) \leq ND_i$. Also assume that, in the presence of an overload, the schedule cannot guarantee the nominal deadline, ND_i, of a general process, P_i. However, an overload only leads to a catastrophic timing failure if P_i ends execution by a time $t > HD_i$. Therefore, the schedule is stable for overloads that lead P_i to complete execution during the time interval $[c(P_i), HD_i]$. Since it is assumed that $GT_i > 0$—that is, $HD_i > ND_i$—$[c(P_i), HD_i]$ is a non-zero time interval. Thus, the stability margin of the schedule is greater than zero. It is worth noting that this conclusion applies to any schedule, independently of the scheduling algorithm from where it results. ■

3.1. Additional Definitions and Assumptions

A few definitions and assumptions must be presented before quantifying the stability of a pre-run-time schedule:

- A schedule that is not disturbed by an overload is said to run under *nominal conditions*. Under nominal conditions, the execution time of a process P_i is considered to be equal to its upper bound, C_I, which can be defined as P_i's *nominal execution time*. When a nominal schedule cannot be met due to the occurrence of an overload, it is said to run under an *overload condition*.

- The *nominal completion* time of a process P_i, denoted as $c(P_i)$, is the time by which P_i completes execution under nominal conditions. In the same way, the *nominal start time* of P_i, denoted as $s(P_i)$, is the time by which P_i starts execution under nominal conditions. As stated, the parameter ΔC_i expresses an increase in P_i's execution time relative to C_i. Therefore, ΔC_i is the *magnitude* of an overload introduced by P_i. The parameter ΔC_i^* denotes the maximum value of ΔC_i that does not cause any timing failure—i.e., no deadline to be missed. Similarly, ΔC_i^{**} denotes the ΔC_i maximum that does not cause any catastrophic timing failure.

- The *nominal laxity* of a process P_i, $nl(P_i)$, is the difference between its nominal deadline and its nominal completion time. That is, $nl(P_i) = ND_i - c(P_i)$. The *critical laxity* of a process P_i, $cl(P_i)$, is the difference between its hard deadline and its nominal completion time: $cl(P_i) = HD_i - c(P_i)$, or $cl(P_i) = ND_i + GT_i - c(P_i) = nl(P_i) + GT_i$.

- The relation $(P_i \Rightarrow P_j)$ denotes that, according to a pre-run-time schedule, process P_j executes after the execution of process P_i. Note that $(P_i \prec P_j)$ imposes $(P_i \Rightarrow P_j)$, while the opposite may not be true.

- The total amount of time that, under nominal conditions, a processor is idle during the interval $[t_1, t_2]$ is denoted as $\varnothing_{t_1}^{t_2}$.

It is assumed that when a feasible pre-run-time schedule runs under an overload condition no processes are skipped and the expected sequence of process executions is kept. These assumptions require a run-time synchronisation mechanism to be added to a pre-run-time schedule. In centralised systems such a mechanism is just a FIFO queue run-time managed by the only processor existing in the computer system. When, due to an overload introduced

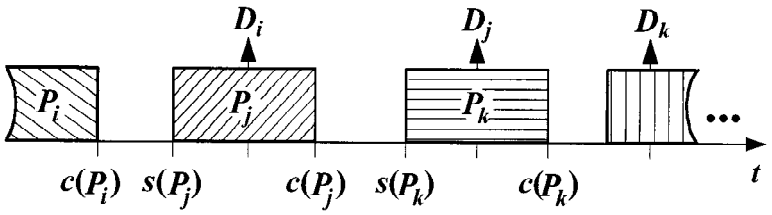

Figure 3. A segment of a general schedule.

by a process P_i, the execution of a process P_j such that $(P_i \Rightarrow P_j)$ cannot be started at the expected time, P_j's execution request is placed at the end of the queue. Process P_j will start execution as soon at its execution request gets the top of the queue and all the requests from processes scheduled to execute before P_j have been satisfied.

A FIFO queue attached to each processor and managed in the way stated in the last paragraph, while necessary, does not suffice for a general multiprocessor system. This is because synchronisation constraints between processes executing on different processors cannot be guaranteed in this way. It is also necessary to include a run-time massage passing system between processors. This mechanism is simple to implement and works in this way:

Every time a processor V_i completes the execution of a process P_i that, due to a synchronisation constraint, is scheduled to execute before a process P_j allocated to a different processor V_j, V_i sends an according message to V_j. Processor V_j only executes P_j after receiving such a message.

The assumption that when a feasible pre-run-time schedule operates under an overload condition no processes are skipped and the sequence of process executions is kept, has two important consequences. First, it means that during an overload no process starts its execution before its request time, or delays its execution by more than the absolutely required. Second, exclusion and precedence constraints are guaranteed. This leads to an important conclusion in the context of the proposed stability criterion:

The only concern about the stability of a feasible schedule relies on guarantying process hard deadlines. This is given by the quantification of the stability margin of a pre-run-time schedule.

3.2. Single Processor System Analysis

The exact characterisation of the impact of a transient overload upon a feasible pre-run-time schedule is central to the establishment of its stability. Single processor systems running single deadline processes are first considered. This means that a general process P_i is viewed has having a deadline D_i and a laxity $l(P_i) = D_i - c(P_i)$.

Consider a segment of a general and feasible schedule where the deadline of the process P_i that introduces an overload as well as the deadlines of processes executing after P_i are arbitrarily placed—figure 3.

First thing to note is that P_i cannot increase its execution time by more than $D_i - c(P_i)$ without missing its deadline. Therefore:

$$\Delta C_i^* \leq l(P_i). \tag{6}$$

On the other hand, P_i cannot increase its execution time by more than $s(P_j) - c(P_i)$ without causing a delay on P_j's start time. Additionally, P_j's completion cannot be delayed by more $l(P_j)$ without causing a timing failure. The maximum tolerable delay on P_j's start time is thus $l(P_j)$. Therefore, one finds that:

$$\Delta C_i^* \leq s(P_j) - c(P_i) + l(P_j). \tag{7}$$

Since during the time interval $[c(P_i), s(P_j)]$ the processor is idle, the amount of time $s(P_j) - c(P_i)$ can be expressed as $\varnothing_{c(P_i)}^{s(P_j)}$. Expression (7) can thus take the form:

$$\Delta C_i^* \leq \varnothing_{c(P_i)}^{s(P_j)} + l(P_j). \tag{8}$$

In the general case, every process P_k executed after P_i cannot delay its start time by more than its nominal laxity, $l(P_k)$, and has its start time delayed by $\Delta C_i - \varnothing_{c(P_i)}^{s(P_k)}$ as a consequence of the overload introduced by P_i. Thus, the condition $\Delta C_i - \varnothing_{c(P_i)}^{s(P_k)} \leq l(P_k)$ has to be satisfied for every process P_k such that $(P_i \Rightarrow P_k)$. Consequently, the stability margin of the schedule for process P_i is given by:

$$\begin{aligned} &\Delta C_i^* = \min\left(l(P_i), \varnothing_{c(P_i)}^{s(P_k)} + l(P_k)\right) \\ &\forall (P_i \Rightarrow P_k) \mid \varnothing_{c(P_i)}^{s(P_k)} < l(P_i), \end{aligned} \tag{9}$$

where,

$$\varnothing_{c(P_i)}^{s(P_k)} = s(P_k) - c(P_i) - \sum_{j=i+1}^{k-1} C_j = s(P_k) - s(P_i) - \sum_{j=i}^{k-1} C_j. \tag{10}$$

An overload occurring during the execution of a process P_i manifests itself during the time interval $[t_s, t_c]$, where t_s and t_c represent *the start* and *the cease time* of the overload, respectively. The t_s value is given by $t_s = c(P_i)$, since nominal scheduling conditions are abandoned at this time. When a process P_i enlarges its execution time by a value ΔC_i, it introduces an overload that will be extinguished as soon as the ΔC_i time is stolen from the nominal idle times. Therefore, t_c is the smallest t time that satisfies the equality:

$$\Delta C_i = \varnothing_{t_s}^{t} = \varnothing_{c(P_i)}^{t}. \tag{11}$$

For a ΔC_i^* magnitude overload t_c is deduced from the expression:

$$\begin{aligned} &\Delta C_i^* = \varnothing_{c(P_i)}^{t_c} = t_c - c(P_i) - \textstyle\sum C_j \\ &\forall P_j \mid \varnothing_{c(P_i)}^{s(P_j)} < \Delta C_i^*. \end{aligned} \tag{12}$$

Remember that the fault-tolerance model has assumed that the effects of two consecutive overloads never overlap. Therefore, the maximum duration of an overload defines the

minimum time interval between two consecutive overloads. Such a time is very important to fault-tolerance, since it defines the maximum frequency of faults the system can handle.

It is important noting that expression (9) establishes the stability margin of a schedule for a particular execution of a process P_i. However, P_i is a general element of a set of periodic processes whose periods are arbitrary. Therefore, a more detailed analysis is required for quantifying the stability of a schedule for a process P_i.

Let L be the least common multiple (LCM) of the periods of the processes executed on a processor. Thus, during a time interval $[t, t + L]$ P_i executes L/T_i times. It must be noted that calculation to LCM is always required to devise a feasible pre-run-time schedule (Locke, 1992). Therefore, calculation to LCM is always feasible if a feasible schedule is assumed.

Let $\Delta C_i^*(j)$ denote the stability margin of the schedule for the jth execution of P_i in the interval $[t, t + L]$. The parameter $\Delta C_i^*(j)$ is established according to expression (9) considering the particular sequence of process executed after the jth execution of P_i. The stability margin of the schedule for process P_i, $\overline{\Delta} C_i^*$, is thus given by:

$$\overline{\Delta} C_i^* = \min\left(\Delta C_i^*(1), \Delta C_i^*(2), \ldots, \Delta C_i^*(L/T_i)\right). \tag{13}$$

THEOREM 2 *A feasible pre-run-time-schedule of processes having non-zero grace time and executed on a single processor system remains stable when a single process P_i increases its execution time by no more than:*

$$\overline{\Delta} C_i^{**} = \min\left(\Delta C_i^{**}(1), \Delta C_i^{**}(2), \ldots, \Delta C_i^{**}(L/T_i)\right), \tag{14}$$

where each $\Delta C_i^{**}(j)$ is the maximum increase in P_i's execution time for its jth execution in the time interval $[t, t + L]$, and is calculated according to the formula:

$$\begin{aligned} &\Delta C_i^{**}(j) = \min\left(cl(P_i), \varnothing_{c(P_i)}^{s(P_k)} + cl(P_k)\right) \\ &\forall (P_i \Rightarrow P_k) \mid \varnothing_{c(P_i)}^{s(P_k)} < cl(P_i). \end{aligned} \tag{15}$$

Proof: The proof for theorem 2 directly follows from the analysis that derived expressions (9) and (13), while considering the hard deadline and the critical laxity of each process. ■

THEOREM 3 *For a set of processes having a grace time greater than zero and executed on a single processor system, the value of ΔC_i^{**} associated to a process P_i is always greater than ΔC_i^*.*

Proof: Note that the equally $cl(P_i) = nl(P_i) + GT_i$ applies for each process P_i. Therefore:

$$\min\left(cl(P_i), \varnothing_{c(P_i)}^{s(P_k)} + cl(P_k)\right) = \min\left(nl(P_i) + GT_i, \varnothing_{c(P_i)}^{s(P_k)} + nl(P_k) + GT_k\right).$$

This is true since processor idle times do not depend on the placement and characterisation of the deadlines. Therefore, and because the grace time of each process is greater than zero,

one finds that:

$$\min\left(nl(P_i) + GT_i, \varnothing_{c(P_i)}^{s(p_k)} + nl(P_k) + GT_k\right) > \min\left(nl(P_i), \varnothing_{c(P_i)}^{s(P_k)} + nl(P_k)\right). \quad \blacksquare$$

Another important conclusion emerges from the presented analysis:

Corollary 1 *In a centralised system executing a P set of n processes scheduled according to a feasible pre-run-time scheduling algorithm, $\Omega(t, P)$, the maximum time redundancy that can be used for error processing is given by:*

$$RT_{\max}(\Omega) = \min\left(\overline{\Delta}C_1^{**}, \overline{\Delta}C_2^{**}, \ldots, \overline{\Delta}C_n^{**}\right). \tag{16}$$

Proof: Corollary 1 is true because a real-time system cannot include a time redundancy that can lead any process to miss its hard deadline during an error processing action. Therefore, $RT_{\max}$ cannot be greater than the stability margin of the schedule for any process. ■

Expression (16) defines the *stability margin* of a pre-run-time schedule where no time redundancy is used for error processing. Therefore, a real-time system that includes a time redundancy $RT \leq RT_{\max}$ for error processing purposes has a stability margin, $\varphi(\Omega)$, given by:

$$\varphi(\Omega) = RT_{\max}(\Omega) - RT. \tag{17}$$

Finally, another important conclusion:

Corollary 2 *The scheduling criterion that must be optimised by a pre-run-time scheduling algorithm intended to maximise the stability of a single processor system is the maximisation of processes' laxity.*

Proof: Note that the nominal laxity of a process P_i increases when a feasible schedule anticipates P_i's nominal start time. Processor idle time between the completion of P_i and the start time of any other process executed after P_i also increases in this situation. Therefore, according to expression (9), the parameter ΔC_i^* increases. On the other hand, if the laxity of a process P_j, such that $(P_i \Rightarrow P_j)$, is increased by anticipating its start time by a value ξ, the value $\varnothing_{c(P_i)}^{s(P_j)}$ decreases by ξ, but the sum $\varnothing_{c(P_i)}^{s(P_j)} + l(P_j)$ keeps the original value. This shows that ΔC_i^* does not depend on P_j's start time. Therefore, when the laxity of a process P_i increases, so does the stability of the schedule for P_i. ■

It is worth noting that the maximisation of a process laxity is equivalent to the maximisation of its critical laxity. However, the adjectives *nominal* and *critical* were intentionally omitted in the statement of corollary 3. This was to emphasise that the optimisation criterion does not need to assume any consideration about processes deadlines. Therefore, any pre-run-time scheduling algorithm that satisfies the laxity maximisation criterion for processes having a single deadline also provides the maximum stability to a schedule of processes having arbitrary grace times.

In (Xu and Parnas, 1990) it is presented a centralised pre-run-time scheduling algorithm that satisfies the optimisation criterion stated above. The algorithm is optimal in the sense that it always finds a feasible schedule providing that such a schedule exists. Another advantage of this algorithm is that it departs from a very general load model similar to ours. Therefore, the Xu and Parnas algorithm must be used in single processor pre-run-time scheduled applications.

3.3. Multiprocessor System Analysis

In a multiprocessor system, a set of processes $P(V_i)$ is said to be independent of the set $P(V_j)$ if no precedence or exclusion relations exist between a process P_i belonging to $P(V_i)$ and a process P_j belonging to $P(V_j)$. A real time system is *composed of independent process sets* if:

$$\forall P(V_i), P(V_j) \mid V_i \neq V_j,$$
$$\neg\exists(P_i, P_j) \mid P_i \in P(V_i) \wedge P_j \in P(V_j) \wedge \big((P_i \prec P_j) \vee (P_j \prec P_i) \vee (P_i \otimes P_j)\big).$$

Due to the absence of synchronisation constraints between processes executing on different processors, a real-time system composed of independent process sets has no *overload propagation paths* between processors. This means that there are no process executing sequences that can make an overload introduced by a process $P_i \in P(V_i)$ to disturb the nominal execution of a process $P_j \in P(V_j)$. Consequently, the analysis developed in the last subsection directly applies for each processor of a multiprocessor system composed of independent process sets.

However, multiprocessor systems composed of independent process sets are rare in practice. This means that most multiprocessor real-time systems have paths through which an overload can propagate from one processor to another. Overload propagation paths can be established with the help of a graph. In such a graph, nodes represent processes, and arcs denote partial process execution orderings. The root node denotes a process that introduces an overload. Nodes are organised in rows. Each row represents a particular processor. An arc connecting two nodes in different rows denote a path from where an overload can propagate from one processor to another.

An *overload propagation graph* is denoted as $G_{OP}(P_i) = (N_p, A_o)$, where P_i is the root node. N_p and A_o denote the set of nodes and the set of arcs, respectively. Figure 4 shows the overload propagation graph for the process P_1 belonging to a multiprocessor real-time system having three processors and seven processes, such that $P(V_1) = \{P_1, P_2, P_3\}$, $P(V_2) = \{P_4, P_5, P_6\}$, $P(V_3) = \{P_7\}$, $(P_1 \prec P_4)$ and $(P_5 \otimes P_7)$. It is assumed that a pre-run-time scheduling algorithm has established a feasible schedule according to the following process execution orderings:

For processor V_1, $\big[P_1 \rightarrow P_2 \rightarrow P_3\big]$; for V_2, $\big[P_4 \rightarrow P_5 \rightarrow P_6\big]$; for V_3, $[P_7]$.

Also considered is that the exclusion constraint $(P_5 \otimes P_7)$ gave rise to the precedence constraint $(P_5 \prec P_7)$. That is, process P_5 is executed before process P_7 according to some

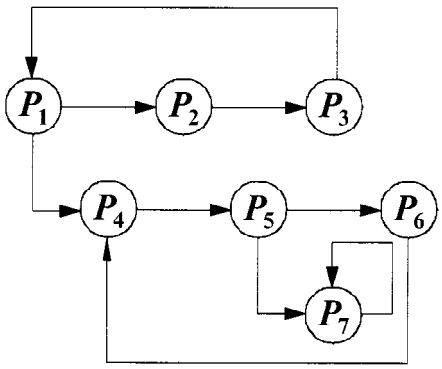

Figure 4. Overload propagation graph for process P_1.

feasible schedule. Using an overload propagation graph its possible to devise the different paths from where an overload introduced by the process denoted as the root node spreads to other processes. Such a set is represented as:

$$OPP(P_i) = \{(P_i, P_j, \ldots, P_k) \mid P_i, P_j, \ldots, P_k \in P \wedge [P_i \rightarrow P_j \rightarrow \cdots \rightarrow P_k]\}. \tag{18}$$

The $OPP(P_i)$ set is central to the stability analysis of a pre-run-time schedule executed on a multiprocessor system. Also important in this context is the analysis developed in the last subsection. This is because the maximum increase in P_i's execution time for a particular propagation path is given by expression (9). Using this expression for all the propagation paths that originate from P_i, one finds the maximum allowable increase in P_i's execution time for each overload propagation path. The most stringent value of this set denotes the maximum increase in P_i execution that does not cause any timing failure. Therefore, for the jth execution of a process P_i having a single an arbitrary deadline, we have:

$$\begin{aligned} &\Delta C_i^*(j) = \min\left((l(P_i), \varnothing_{c(P_i)}^{s(P_k)} + l(P_k)\right), \\ &\forall\left([P_i \rightarrow \cdots \rightarrow P_k] \in OPP(P_i)\right) \mid \varnothing_{c(P_i)}^{s(P_k)} < l(P_i). \end{aligned} \tag{19}$$

Since P_i is a periodic process, one concludes that:

$$\overline{\Delta} C_i^* = \min\left(\Delta C_i^*(1), \Delta C_i^*(2), \ldots, \Delta C_i^*(L/T_i)\right), \tag{20}$$

where L is the least common multiple of the periods of the processes allocated to the processor on which P_i executes, and $\Delta C_i^*(j)$ is the ΔC_i^* value for the jth execution of P_i in the time interval $[t, t + L]$.

From here, it is possible to establish a set of theorems similar to those developed for single processor systems while considering a multiprocessor real-time system.

THEOREM 4 *A feasible pre-run-time-schedule of processes having non-zero grace time and executed on a multiprocessor system remains stable when a single process P_i increases its execution time by no more than:*

$$\overline{\Delta}C_i^{**} = \min\left(\Delta C_i^{**}(1), \Delta C_i^{**}, (2), \ldots, \Delta C_i^{**}(L/T_i)\right), \tag{21}$$

where each $\Delta C_i^{**}(j)$ is the maximum increase in P_i's execution time for its jth execution in the time interval $[t, t+L]$, and is calculated according to the formula:

$$\begin{aligned} &\Delta C_i^{**}(j) = \min\left(cl(P_i), \varnothing_{c(P_i)}^{s(P_k)} + cl(P_k)\right), \\ &\forall\left(\left[P_i \rightarrow \cdots \rightarrow P_k\right] \in OPP(P_i)\right) \mid \varnothing_{c(P_i)}^{s(P_k)} < cl(P_i). \end{aligned} \tag{22}$$

Proof: The proof for theorem 4 directly follows from the analysis that derived expressions (19) and (20), while considering the hard deadline and the critical laxity of each process. ■

THEOREM 5 *For a set of processes having a grace time greater than zero and executed on a multiprocessor system, the value of ΔC_i^{**} associated to a process P_i is always greater than ΔC_i^{*}.*

Proof: The approach is similar to the proof of theorem 3, while considering the values of ΔC_i^{*} and ΔC_i^{**} in the context of a multiprocessor system. ■

Corollary 3 *For a system given by a V set of m processors and a P set of n processes feasibly scheduled according a pre-run-time scheduling algorithm, $\Omega(t, P, V)$, the maximum time redundancy that can be used for error processing in a $V_i \in V$ processor is given by:*

$$\begin{aligned} &RT_{\max}(\Omega, V_i) = \min\left(\overline{\Delta}C_1^{**}, \overline{\Delta}C_2^{**}, \ldots, \overline{\Delta}C_n^{**}\right) \\ &\forall P_i \in P(V_i). \end{aligned} \tag{23}$$

Proof: The proof for corollary 3 is similar to the proof for corollary 1. In this case it should be noted that $RT_{\max}(\Omega, V_i)$ cannot be greater than the stability margin of the schedule running in V_i for any process $P_i \in P(V_i)$. ■

It is worth noting that the stability margin for a multiprocessing system based on a pre-run-time schedule has *m* components. Every component is given by expression (23). Thus, the stability margin of a processor V_i that uses a time redundancy $RT(V_i)$ for error recovery and belongs to a multiprocessing system is given by:

$$\varphi(\Omega, V_i) = RT_{\max}(\Omega, V_i) - RT(V_i). \tag{24}$$

The stability margin of a multiprocessing pre-run-time schedule of a P set of processes, $\Omega(t, P)$, is thus given by the following vector:

$$\varphi(\Omega) = \begin{bmatrix} \varphi(\Omega, V_1) \\ \varphi(\Omega, V_2) \\ \vdots \\ \varphi(\Omega, V_m) \end{bmatrix}. \tag{25}$$

Corollary 4 *The scheduling criterion that must be optimised by a pre-run-time scheduling algorithm intended to maximise the stability of a multiprocessor system is the maximisation of processes' laxity.*

Proof: The proof directly follows from the proof of corollary 2. In this case, P_i is the process represented by the root node of an overload propagation path, and P_j is a process represented by an arbitrary node on the same path. ■

There is a multiprocessing pre-run-time scheduling algorithm that maximises processes' laxity. This algorithm was developed by Shepard and Cagné (1991) and derives from that developed by Xu and Parnas for single processor systems. The Shepard and Cagné algorithm is claimed to be optimal for multiprocessor pre-run-time scheduling, and uses a very general load model similar to ours. Therefore, the Shepard and Cagné algorithm must be considered in multiprocessor pre-run-time scheduled systems.

3.4. Multiple Processes Re-execution

Rolling back solely the executing process every time an error is detected is only effective for processes that execute for completion when their requests are satisfied. Otherwise, the executing process as well as all the preempted processes must rollback their executions every time an error is detected (Randell, 1975). This means that the stability analysis developed so far only applies to non-preemptive pre-run-time scheduling. However, non-preemptive real-time scheduling is avoided whenever possible: it is a NP-hard problem (Cheng, Stankovic and Ramamrithan, 1987) and usually produces inefficient schedules. Therefore, there is a strong interest in adapting to preemptive schedules the stability analysis already developed.

The preemptive schedule of a P set of n processes takes the segmentation of each process P_i into a convenient number, $z(i) \geq 1$, of segments: $P_{i,1}, P_{i,2}, \ldots P_{i,z(i)}$. This means that when the jth $\neq z(i)$ segment of a process P_i completes execution, P_i is temporarily suspended. P_i resumes execution by the time its $(j+1)$th segment starts execution. Since process segmentation and multiple processes re-execution are now considered, the scheduling model used so far has to be slightly broadened. The following assumptions are considered from now on:

1. The segmentation of a process P_i into $z(i) \geq 1$ parts, gives rise to segments $P_{i,1}, P_{i,2}, \ldots, P_{i,z(i)}$, such that: $\left(P_{i,1} \prec P_{i,2}\right), \left(P_{i,2} \prec P_{i,3}\right), \ldots, \left(P_{i,z(i)-1} \prec P_{i,z(i)}\right)$.

2. A pre-run-time schedule is feasible for a process P_i if:

   - Segment $P_{i,1}$ does not start execution before P_I's request time;
   - Precedence and exclusion constraints between processes segments are respected;
   - Segment $P_{i,z(i)}$ does nor complete execution after P_i's nominal deadline, ND_i.

3. A feasible pre-run-time schedule is stable for a process P_i if segment $P_{i,z(i)}$ does not complete execution after P_i's hard deadline when an error processing action takes place.

4. Every process P_i contains an arbitrary number of checkpoints that are arbitrarily placed on P_i execution code. However, it is assumed that:
   - Every process P_i performs an acceptance test by the end of the execution of its $P_{i,z(i)}$ segment. If the test passes, the execution of P_i is declared completed. If it fails, P_i rollbacks to the last recovering point, and restarts execution from there. This avoids the need of re-executing any segment of P_i after P_i's completion.
   - If the segment $P_{i,k}$ of a process P_i is executed on a processor V_i, and the segment $P_{i,k+1}$ is executed on a different processor V_j, then the segment $P_{i,k}$ ends with an acceptance test. This avoids the need of re-execution processes allocated to multiple processors during an error recovering action.
5. Every time an error is detected the executing process and all the preempted processes rollback execution to their last recovery points.

According to assumptions 2 and 3, the feasibility and the stability analysis of a preemptive pre-run-time schedule does not require the consideration of the nominal and hard deadline of a process segment $P_{i,k}$, for $k \neq z(i)$. However, since processes segments are supposed to have characteristics similar to those defined in the load model, it can be stated that:

$$ND_{i,1} = ND_{1,2} = \cdots = ND_{i,z(i)} = ND_i, \tag{26}$$

and

$$HD_{i,1} = HD_{1,2} = \cdots = HD_{i,z(i)} = HD_i. \tag{27}$$

Consider thus a pre-run-time scheduling segment $P_{k,k} \rightarrow \cdots \rightarrow P_{j,j} \rightarrow P_{i,i}$, such that an error detected during $P_{i,i}$ execution can only be properly recovered by re-executing segments $P_{k,k}, \ldots, P_{j,j}, P_{i,i}$. $P_{i,i}$ is a general segment of process P_i; namely, it can refer to its last segment, $P_{i,z(i)}$. The condition required for processing such an error without causing a catastrophic timing failure is given by the following corollary:

Corollary 5 *It is possible to process an error detected during the execution of a segment $P_{i,i}$ by re-executing the segments $P_{k,k}, \ldots, P_{j,j}, P_{i,i}$, if it does not require a time greater than $\Delta C^{**}_{i,i}$.*

Proof: Note that $\Delta C^{**}_{i,i}$ is calculated using expressions (15) or (22) as it relates to a single or multiprocessor system, respectively. Remember that $\Delta C^{**}_{i,i}$ defines the maximum increase in $P_{i,i}$ execution time that does not lead to a catastrophic time failure. However, $\Delta C^{**}_{i,i}$ can also be defined as the maximum delay on $P_{i,i}$ completion time that does not lead $P_{i,i}$ or *any process segment executed after it* to miss its hard deadline. This delay can have any cause. Namely, the re-execution of segments $P_{k,k}, \ldots, P_{j,j}, P_{i,i}$.

Also important to note is that the need of re-executing the segments $P_{k,k}, \ldots, P_{j,j}$ as a consequence of an error detected during $P_{i,i}$ execution is synonymous that segments $P_{k,z(k)}, \ldots, P_{j,z(j)}$ *are executed after* $P_{i,i}$. Otherwise, there was no need to re-execute them, since processes $P_k, \ldots, P_j$ were already completed by the time the error is detected. Thus, corollary 5 gives the condition for segment $P_{i,i}$ as well as segments $P_{k,z(k)}, \ldots, P_{j,z(j)}$ not to miss their hard deadlines. Consequently, if defines the condition for using backward error recovery in a preemptive pre-run-time schedule. ■

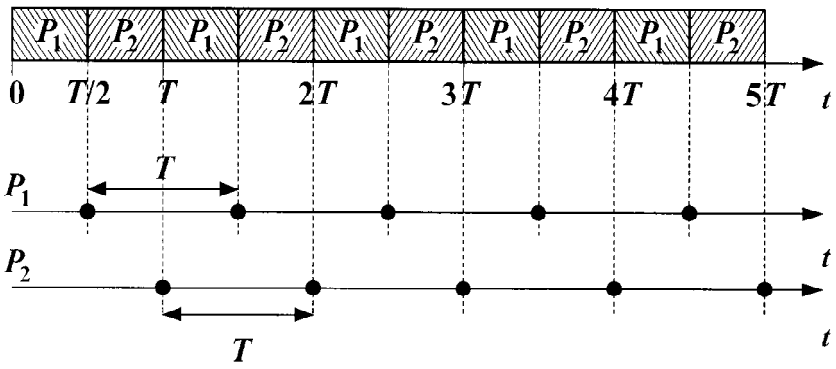

Figure 5. A saturated schedule.

4. A Final Note on System Recovery

According to expression (11), the cease time of an overload depends on nominal idle times following the start time of the overload. Therefore, it can be concluded that a *saturated system*—that is, a system where nominal processor idle times do not exist—never returns to nominal conditions after suffering the impact of an overload.

However, this view does not apply to many real-time applications. Consider the scenario depicted in figure 5. It represents the only feasible schedule for a centralised real-time application consisting of two periodic processes, P_1 and P_2 requesting execution every time $t = kT$ and $t = (2k + 1) * T/2$, respectively, for $k \geq 0$. It is assumed that $T_1 = T_2 = T$, $C_1 = C_2 = T/2$, $ND_1 = ND_2 = T/2$ and $HD_1 = HD_2 = T$. Also presented in figure 5 is the *observation grid* for processes P_1 and P_2. An observation grid is the set of points associated to the completion time of a process P_i (Kopetz, 1991). Observation points are represented by bold dots. Let $\Omega_N(t, P_1, P_2)$ denote the nominal schedule represented in figure 5.

Consider now that the execution of P_2 starting at the time $t = 3T/2$ introduces an overload such that $\Delta C_2 < T/2$. The impact of this overload upon process sequencing is represented in figure 6. The dashed dots represent the nominal observation points. Let $\Omega_O(t, P_1, P_2)$ denote the schedule represented in figure 6. First thing to note is that $\overline{\Delta}C_1^{**} = \overline{\Delta}C_2^{**} = \min(cl(P_1), cl(P_2)) = T/2$. Since $\Delta C_2 < T/2$, the overload does not cause any catastrophic timing failure. On the other hand one finds that:

$$\Omega_O(t, P_1, P_2) = \begin{cases} \Omega_N\big(t, P_1, P_2\big) & \text{for } t \leq 2T; \\ \Omega_N\big((t - \Delta C_2), P_1, P_2\big) & \text{for } t > 2T; \end{cases} \tag{28}$$

This means that nevertheless the nominal deadlines of both processes can no longer be meet for a time $t \geq 2T$, the period of each observation grid recovers its nominal value, T, a short time after the occurrence of the overload. Therefore, if the role of both processes, P_1 and P_2, is to establish an observation grid with a period T, one may state that processes P_1 and P_2 recover their nominal conditions at the times $5T/2 + \Delta C_2$ and $2T + \Delta C_2$,

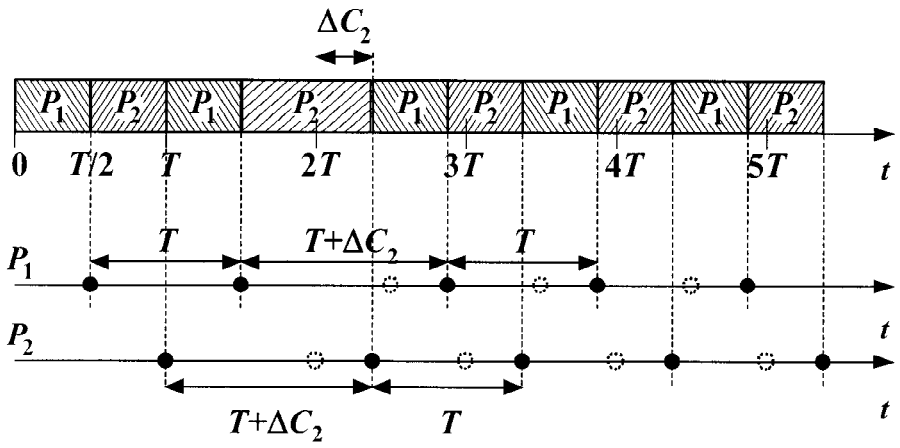

Figure 6. A saturated schedule under an overload condition.

respectively. Thus, it can be argued that both processes miss a single nominal deadline due to the occurrence of the overload. However, each observation grid has changed its phase by ΔC_2 time units as a consequence of the overload.

When a similar analysis is derived for the observation grids attached to a set of arbitrary periodic processes, one finds that if a process P_i increases its execution time by a margin $\Delta C_i \leq \overline{\Delta C}_i^{**}$, then all processes making part of the system have, at most, one timing failure. This is because no more than two observation points for a process P_j become separated by more than ND_j as a consequence of the overload. On the other hand, the phase change exhibited by an observation grid after suffering the impact of an overload is always lesser than or equal to ΔC_i. This is because processor nominal idle times contribute for the returning of a grid to its nominal observation points. However, idle times are not necessary for making an observation grid to return to its nominal period.

We feel this note is important because, in many systems, the major real-time issue is guaranteeing that two consecutive executions of a process are not separated by more than a certain time. This is particularly true in control applications (Middleton and Goodwin, 1990; Åstrom and Wittenmark, 1990), where the execution of a periodic process is the way for enforcing a timing regularity in the observation of the controlled object according to its dynamic. A periodic control process typically reads data from the application environment, performs data manipulations, and writes results on an output port, changing the environment accordingly (Stankovic and Ramamritham, 1993). For a process P_i that performs this way, time is a *relative measure*, in the sense that it restarts counting every time P_i completes an execution. Therefore, the way that P_i recovers from an overload condition must be viewed in the context of *time as a relative quantity*.

While this concept is applicable to many real-time systems, it must be taken very carefully. It can become very dangerous if generalised. Namely, to very critical systems where alarm signals are expected to trigger a very fast operational change; e.g., a safe shutdown or

the reconfiguration of the controlling system. Real-time processes are very diversified. Some of them—including the most critical ones—do not understand time as a relative quantity.

5. Conclusions

The paper has dealt with the stability of pre-run-time schedules executed on single and multiprocessor real-time systems. The point of departure was that most real-time processes have a nominal and a hard deadline separated by a non-zero grace time. Therefore, processes were considered to be scheduled to meet their nominal deadlines under nominal conditions, and not to miss the hard deadlines in the presence of an overload.

The first major contribution of the paper is the proof that the stability of any real-time system scheduled in this way is greater than the stability achieved when it is tuned to guarantee a single and hard deadline for each process. This is important because it shows that error masking is not the only solution for designing highly dependable hard real-time systems. Backward error recovery or another time consuming error processing technique can be used in critical real-time applications when certain conditions are satisfied. The devising of these conditions for a set of various and realistic scenarios is the second major contribution of the paper. The statement of the scheduling criterion that brings the maximum stability to a schedule and the reference to existing algorithms that conform to such a criterion is another important contribution. Therefore, this paper has provided a framework that can provide a great help in the design of low cost and highly dependable pre-run-time scheduled real-time systems.

References

Åström, K., and Wittenmark, B. 1990. *Computer-Controlled Systems. Theory and Design*. 2nd Edn. Prentice-Hall International Editions.

Audsley, N., and Burns, A. 1992. *Real-Time Scheduling*. Department of Computer Science, University of York, U.K. Tech. Report YCS 134.

Avizienis, A. 1997. Toward systematic design of fault-tolerant systems. *IEEE Computer* 30(4): 51–58.

Bond, P., Seaton, D., Veríssimo, P., and Waddington, J. 1991. Real-time concepts. In D. Powell (ed.), *Springer-Verlag Research Reports ESPRIT Series: Delta-4: a Generic Architecture for Dependable Distributed Computing*. Springer-Verlag.

Burns, A. 1991. Scheduling hard real-time systems: a review. *Software Eng. J.*, pp. 116–128.

Burns, A., and Fohler, G. 1991. Incorporating flexibility into offline scheduling for hard real-time systems. Esprit Bra Project 3092: Predictably Dependable Computing Systems. Second Year Report, Vol. 1, Chap. 3, Pt II.

Buttazzo, G., Spuri, M, and Sensini, F. 1995. Value vs. deadline scheduling in overload conditions. *Proc. of the IEEE Real-Time Systems Symp.*, pp. 90–99.

Carlow, G. 1984. Architecture of the space shuttle primary avionics software system. *Comm. ACM* 27(9): 926–936.

Carpenter, T, Driscoll, K., Hoyme, K., and Carciofini, J. 1994. ARINC 659 scheduling: problem definition. *Proc. of the IEEE Real-Time Systems Symp*, pp. 165–169.

Cheng, S-C, Stankovic, J., and Ramamritham, K. 1987. Scheduling algorithms for hard-real time systems: a brief survey. In J. Stankovic and K. Ramamritham (eds.), *Hard Real-Time Systems*. IEEE Computer Society Press, pp. 150–173.

Driscoll, K., and Hoyme, K. 1992. The airplane information system: an integrated real-time flight-deck control system. *Proc. of the IEEE Real-Time Systems Symp.*, pp. 267–270.

Gheith, A., and Schwan, K. 1993. CHAOSarc: kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications. *ACM Trans. on Computer Systems* 11(1): 33–72.

Iyengar, S., Prasad L., and Min, H. 1995. *Advances in Distributed Sensor Technology*. New Jersey: Prentice Hall PTR.

Jensen, E., Locke, C., and Tokuda, H. 1985. A time-driven scheduling model for real-time operating systems. *Proc. of the 1995 IEEE Real-Time Systems Symp.*, pp. 112–122.

Jensen, E.D. 1993. *Asynchronous decentralized realtime computers*. Realtime Computer Systems, Digital Equipment Corp.

Jensen, E. D. 1994. Eliminating the hard/soft real-time dichotomy. *Embedded Systems Programming* 7(10): 28–34.

Kirrmann, H. 1987. Fault tolerance in process control: an overview and examples of European products. *IEEE Micro* 7(5): 27–50.

Kligerman, E., and Stoyenko, D. 1986. Real-time Euclid: a language for reliable real-time systems. *IEEE Trans. on Software Eng.* SE-12(9): 941–949.

Kopetz, H. et al. 1989. Distributed fault-tolerant real-time systems: the mars approach. *IEEE Micro*, pp. 25–40.

Kopetz, H. 1991. Event-triggered versus time-triggered real-time systems. *Proc. Int. Workshop on Operating Systems of the 90s and Beyond*. In A. Karshmer and J. Nehmer (eds.), *Springer-Verlag Lecture Notes in Computer Science*, Berlin, Germany, 563: 87–101.

Kopetz, H. 1995. Why time-triggered architectures will succeed in large hard real-time systems. *Proc. of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Computer Systems*, pp. 2–9.

Kopetz, H., and Veríssimo, P. 1993. Real time and dependability concepts. In Sape Mullender (ed), *ACM Press Frontier Series: Distributed Systems*, 2nd edition. Addison-Wesley, ACM Press.

Laplante, P. 1993. *Real-Time Systems Design and Analysis*. IEEE Computer Society Press.

Laprie, J-C. 1991. Dependability concepts. In D. Powell (ed.), *Springer-Verlag Research Reports ESPRIT Series: Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag.

Laprie, J., Arlat, J., Béounes, C., and Kanoun, K. 1990. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *IEEE Computer*, July, pp. 39–51.

Lehoczky and Ramos-Thuel. 1992. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. *Proc. of the IEEE Real-Time Systems Symp.*, pp. 110–123.

Leveson, N. 1986. Software safety: why, what, and how. *ACM Computing Surveys* 18(2): 125–163.

Liu, C., and Layland, J. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20(1): 46–61.

Locke, C. 1986. Best Effort Decision Making for Real-Time Scheduling. PhD. Thesis. Carnegie-Mellon University.

Locke, C. 1992. Software architectures for hard real-time applications. *Real-Time Systems* 4(1): 37–53.

Magalhães, A. 1995. Estabilização dos Controladores de Tempo-Real Através da Complacência Temporal dos Objectos Controlados. PhD Thesis. Faculty of Engineering, University of Porto, Portugal. (In Portuguese).

Magalhães, A., Rela, M., and Silva, J. 1996. On the nature of deadlines. *Microprocessors and Microsystems* 20(2): 79–88.

Magalhães, A. P. 1996. A survey on estimating the timing constraints of hard real-time systems. *Design Automation for Embedded Systems* 1(3): 213–230.

Middleton, R., and Goodwin, G. 1990. *Digital Control and Estimation: A Unified Approach*. Prentice-Hall International Editions.

Mok, A. 1984. The design of real-time programming systems based on process models. *Proc. of the IEEE Real-Time Systems Symp.*, pp. 5–16.

Nelson, V., and Carroll, B. 1987. *Tutorial: Fault-Tolerant Computing*. IEEE Computer Society Press.

Ramamritham, K. 1993. Real-time databases. *Distributed and Parallel Databases* 1: 199–226.

Ramos-Thuel and Lehoczky. 1993. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. *Proc. of the IEEE Real-Time Systems Symp.*, pp. 160–171.

Randell, B. 1975. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Eng.* SE-1(2): 220–232.

Rennels, D. 1984. Fault-tolerant computing—concepts and examples. *IEEE Trans. on Computers* C-33(12): 1116–1129.

Sha, L. Lehoczky, J., and Rajkumar, R. 1986. Solutions for some practical problems in prioritized preemptive scheduling. *Proc. of the IEEE Real-Time Systems Symp.*, pp. 181–191.

Shepard, T., and Gagné, J. 1991. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. on Software Eng.* 17(7): 669–677.

Shin, G., Krishna, C., and Lee, Y.-H. 1985. A unified method for evaluating real-time computer controllers and its application. *IEEE Trans. on Automatic Control* AC-30(4): 357–366.

Somani, A., and Vaidya, N. 1997. Understanding fault-tolerant and reliability. *IEEE Computer* 30(4): 45–50.

Stankovic, J., and Ramamritham, K. 1993. *Advances in Real-Time Systems*. IEEE Computer Society Press.

Stankovic, J., Spuri, M., Di Natale, M., and Buttazzo, G. 1995. Implications of classical scheduling results for real-time systems. *IEEE Computer* 28(6): 16–25.

Xu, J., and Parnas, D. 1990. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Eng*. 16(3): 360–369.

Xu, J., and Parnas, D. 1991. On satisfying timing constraints in hard-real-time systems. *Proc. of the ACM SIGSOFT '91 Conference on Software for Critical Systems.* New Orleans, Louisiana, pp. 132–146.

Ziv, A., and Bruck, J. 1997. An on-line algorithm for checkpoint placement. *IEEE Trans. on Computers* 46(9): 976–985.